4 Feedforward Neural Networks, Binary XOR, Continuous XOR, Parity Problem and Composed Neural Networks.

4.1 Objectives

The objective of the following exercises is to get acquainted with the inner working of the feed-forward neural network. This simple structure is probably the most popular version in use nowadays, notably in system control and classification applications. But it is not a black box that will simply learn from the presented examples: the learning environment has to be carefully controlled to make it work. But even then success is not guaranteed! It has been noted that large monolithic networks (i.e. large networks that are trained in one pass) as commonly occur in biology can in electronics still suffer from what is called "catastrophic forgetting" or "unlearning". Therefore we will see how many small networks that each are learnt successfully can be assembled into a large network and subsequently post-trained without unlearning. This opens the road to the systematic development of intelligent systems.

4.2 Literature

In order to be able to solve the exercises, consult the following resources:

- Brief Introduction to Neural Networks.
- Complete Guide of Joone (Java Object Oriented Neural Engine).
- A general neural network written in Java, *GNet.java*.
- A zip-file containing Javadocs for all classes in Joone.

4.3 Home assignments

Read the above-named literature so that you are answer the following questions:

- What is the difference of *Single-layer* and *Multilayer* Feedforward Neural Networks?
- What is *supervised learning*?
- Explain the following terms: *epoch*, *training data* and *pattern*.
- How do you usually split the data set into training, validation and testing sets?
- What is the *back-propagation* learning algorithm? Explain it briefly.
- Write the generalized delta rule and explain the terms: *learning rate*, *learning mode* and *momentum*.

Acquaint yourself with the user manual of the neural network simulator Joone. In order to do that, a demonstration of the capabilities of Joone by means of an XOR circuit is given in Appendix A. Please take your time to go through this demonstration using the software as installed on your laboratory computer!

Do the demonstration on the Parity Problem as appended to this text (Appendix B). This gives you some basic skills for doing the experiments composed Neural Networks.

It is faster (and more accurate) to use the provided Java-class *GNet.java* to accomplish the assignments in 4.4.1 and 4.4.2. The GUI may be used to solve all assignments throughout this lab, but for some strange reason it does not really work for composed neural nets! You may need to

write your own Java code to train and test a composed neural network. The Joone complete guide provides you with good examples and hints.

4.4 Lab assignments

4.4.1 The set-theoretic OR

The OR circuit is a digital instantiation of the more general function F= I1+I2-I1.I2, where both the inputs and the output carry values in (0…1). In the following we will study the training of this function in more detail.

1. Change the input file used for the Binary XOR to a set of input/outputs that describe the OR on the value range between 0 and 1 with steps of 0.1. The transition with the true and false output can be placed somewhere in the middle. The value pairs should be randomly ordered before fed to the network. Complete the table below with training error and network behavior when tested. **Remember to reset the weights of the network before each training** (see Hints at the end of Appendix A)

Epochs	*Learning Rate*	*Momentum*	*RMSE*	*Behaviour?*
1000	*0.8*	*0.3*		
2000	*0.8*	*0.3*		
3000	*0.8*	*0.3*		
5000	*0.8*	*0.3*		

Observation:

2. Split this set into a training set and a test set. Describe this division and argue which considerations have led to your choice. Then train the OR again, verify the generalization capability and test the performance. It may be needed to try other divisions to achieve a learning result of sufficient quality. Is the learning time (i.e. epochs) higher, equal or lower? Explain!

Size of Training Set	*Epochs*	*Learning Rate*	*Momentum*	*RMSE*	*Behaviour?*
20 out of 121	*1000*	*0.8*	*0.3*		
	2000	*0.8*	*0.3*		
	5000	*0.8*	*0.3*		
100 out of 121	*1000*	*0.8*	*0.3*		
	2000	*0.8*	*0.3*		
	5000	*0.8*	*0.3*		

Observation:

3. Vary the learning rate between 0.1 and 0.9. Select what you judge is a good compromise between learning speed and quality. Explain your reasoning! Show a plot of learning rate versus the training error. **Use always 5000 epochs for training!**

Learning Rate	*Momentum*	*RMSE*
0.1	*0.3*	
0.3	*0.3*	
0.5	*0.3*	
0.7	*0.3*	
0.9	*0.3*	

Observation:

4. Vary the momentum between 0.1 and 0.9. Keep the best learning rate obtained in the previous exercise. Select what you judge is a good compromise between learning speed and quality. Explain your reasoning! Show a plot of momentum versus training error. **Use always 5000 epochs for training!**

Learning Rate	*Momentum*	*RMSE*
	0.1	
	0.3	
	0.5	
	0.7	
	0.9	

Observation:

5. Set the range from which random values are taken to initialize the weights to 0.1, 0.3 and 0.5 respectively. Use the best combination of learning rate and momentum. How does this influence the learning?

Epochs	*Learning Rate*	*Momentum*	*RMSE*

Observation:

4.4.2 Distance Function

So far learning has almost seemed trivial. This is because the example function is a simple linear one, where a single line can separate the 'good' from the 'bad' examples. In the history of neural networks, Marvin Minsky from the MIT Artificial Intelligence Labs has almost brought the concept to death when he demonstrated in 1969 that the XOR function couldn't be trained on a linear feed-forward network. He was only partially right, but it took till the late eighties before the confidence was restored. This XOR circuit is a digital instantiation of the more general distance function $DF=(I_1-I_2)^2$. In the following we will see how right he was before we prove him wrong.

1. Change the input file used for the Binary XOR to a set of input/outputs that describe DF on the value range between 0 and 1 with steps of 0.1. Split this set into a training set and a test set. Then train the XOR again, verify the generalization capability and test the performance. Take the learning rate at 0.8 and the momentum at 0.1. Now compare the learning behavior with what you have experienced for the OR, and give an explanation

Train. Patterns	*Epochs*	*RMSE*	*Behviour?*
20 / 121			
60 /121			
121 /121			

Observation:

2. Vary the learning rate and the momentum. What are the best settings? Argue what the best remaining error in training the XOR function can be!

Epochs	*Learning Rate*	*Momentum*	*RMSE*

Observation:

4.4.3 Composed Neural Networks

1. For starters we are going to create a network containing the OR function and one with the AND function with a similar continuous value range as above. Split the example sets into a training set and a test set. Describe this division and argue which considerations have led to your choice.

Epochs	*Learning Rate*	*Momentum*	*RMSE*

Observation:

2. Then these networks are combined over a third network and the total is trained for a DF function, using the same training set as in 4.4.2. Compare the training time of this composed network to the one for the monolithic function.

Epochs	*Learning Rate*	*Momentum*	*RMSE*

Observation:

3. The knowledge within the composed network may easily disappear upon subsequent learning. So you are kindly requested to re-do the experiment for low learning rates. Check whether this has made any difference.

Observation:

Epochs	*Learning Rate*	*Momentum*	*RMSE*

4. Now take your optimally trained composed DF network and continue training but this time for a NOR function with continuous value range. What do you observe?

Epochs	*Learning Rate*	*Momentum*	*RMSE*

Observation:

5. And, at the end of this little experiment, lets try to return to where we started from by continuing the training with the DF example set. Is this faster or slower than before?

Epochs	*Learning Rate*	*Momentum*	*RMSE*

Observation:

Appendix A.

Simple XOR

This appendix will guide you through different steps to construct a neural network that solves the classical (binary) XOR problem. A binary XOR has the following truth table:

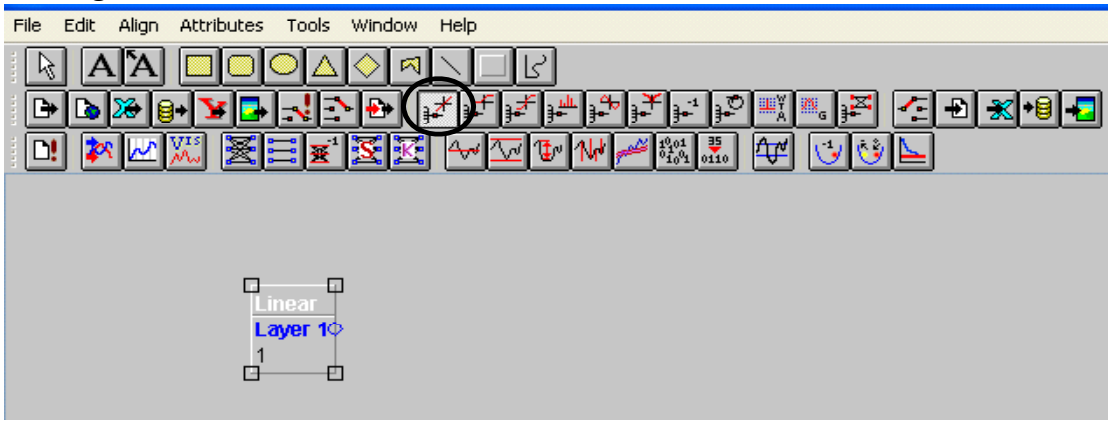
Input 1	Input 2	Output
0	0	0
0	1	1
1	0	1
1	1	0

This table has to be saved in a plaintext file (call it 'binaryXOR_truth_table.txt'). The file contains 4 rows; each with 3 numbers separated by a semi colon ';', as shown below. The numbers may be integer or real.

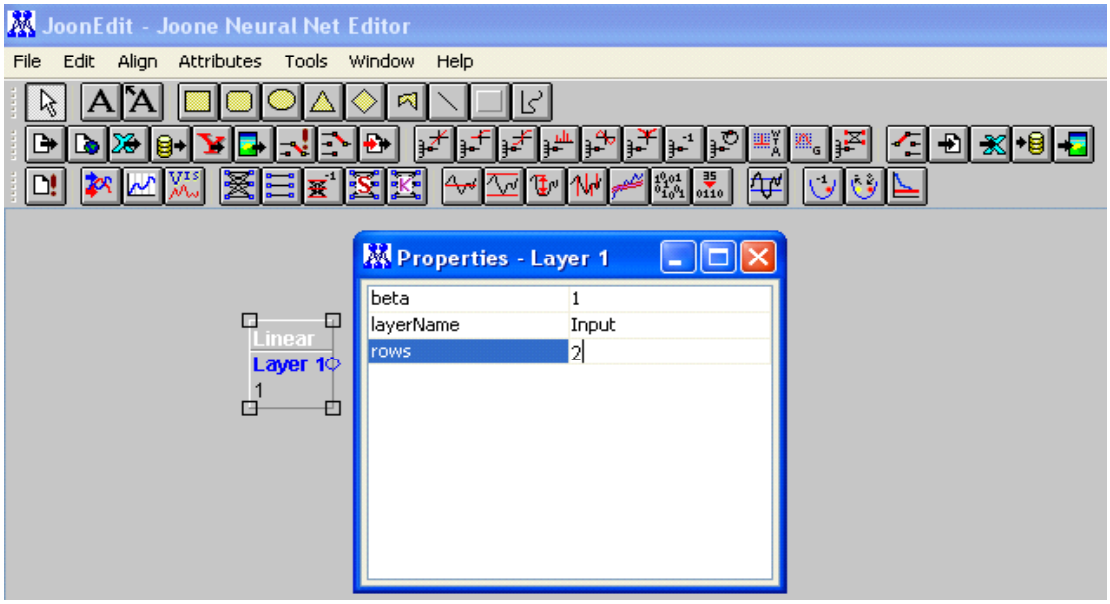
```
0;0;0
0;1;1
1;0;1
1;1;0
```

Now run the Joone GUI editor and follow the steps as described below.

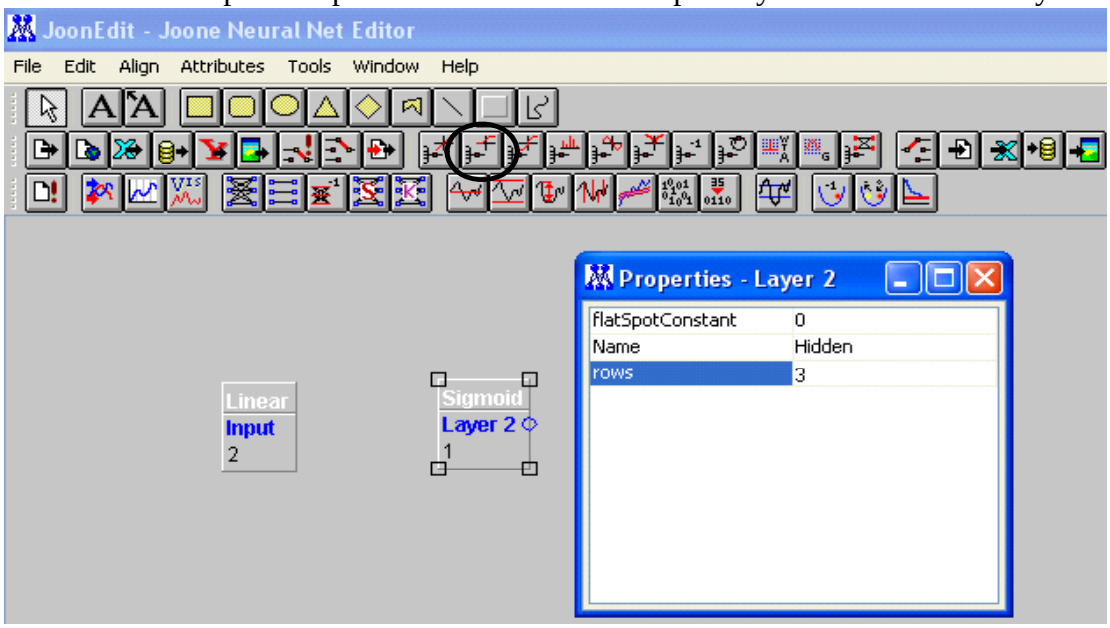
1. Add a *Linear layer* by selecting the encircled button in the figure and then clicking in the drawing area.



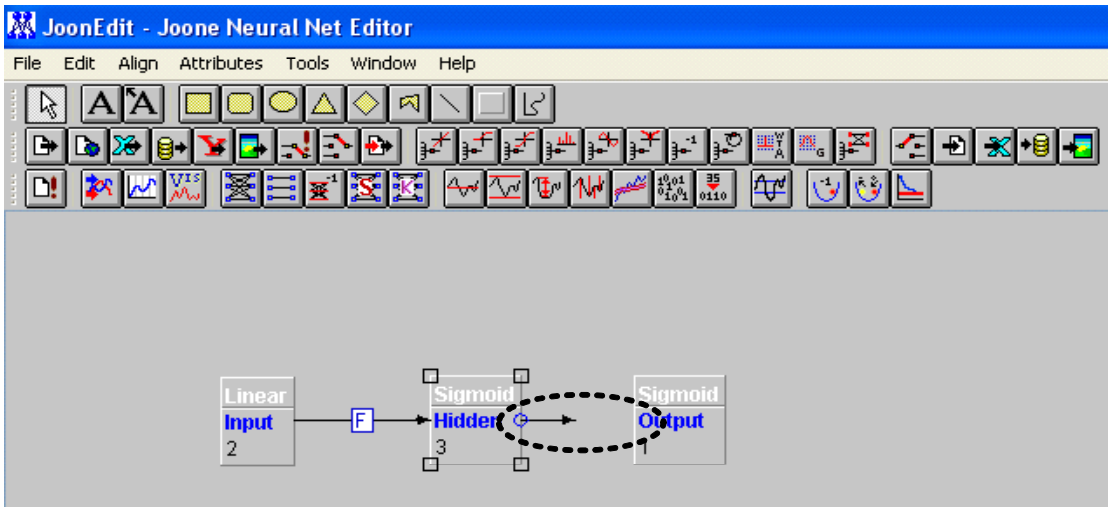
2. Change the name of the layer and the number of neural nodes by viewing the properties (right-mouse click)



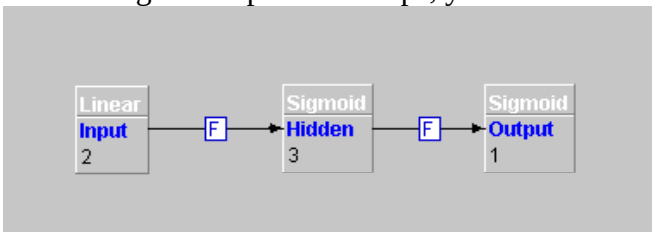
3. Add a new Sigmoid layer by selecting the button marked with a circle in the figure and then clicking in the drawing area. Change the name to ‘Hidden’ and the number of nodes to 3 as shown below. Repeat the procedure and add an ‘Output’ layer with one node only.



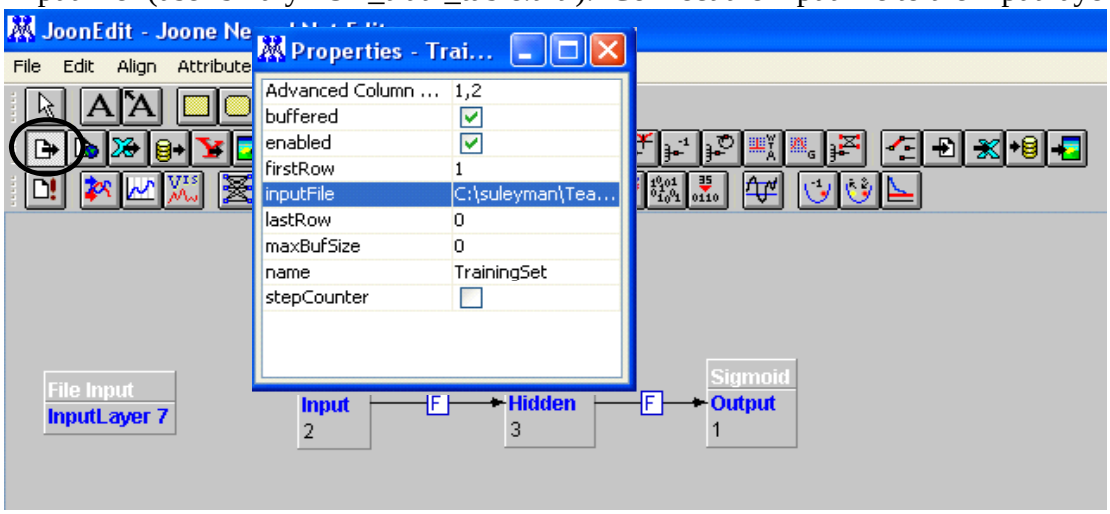
4. Now the three layers are connected to construct a neural network. As each node in a layer has to be connected to all the nodes in the next layer, two ‘Full Synapse’ should be added. This is accomplished by dragging a line from the little circle on the right hand side of a layer and releasing the mouse button when the pointer is on the next layer.



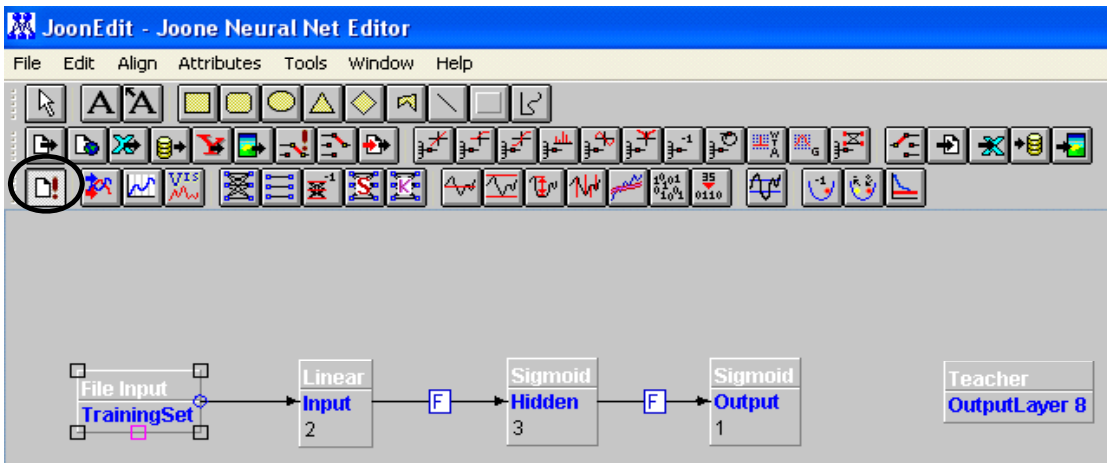
5. After doing all the previous steps, you should have something like:



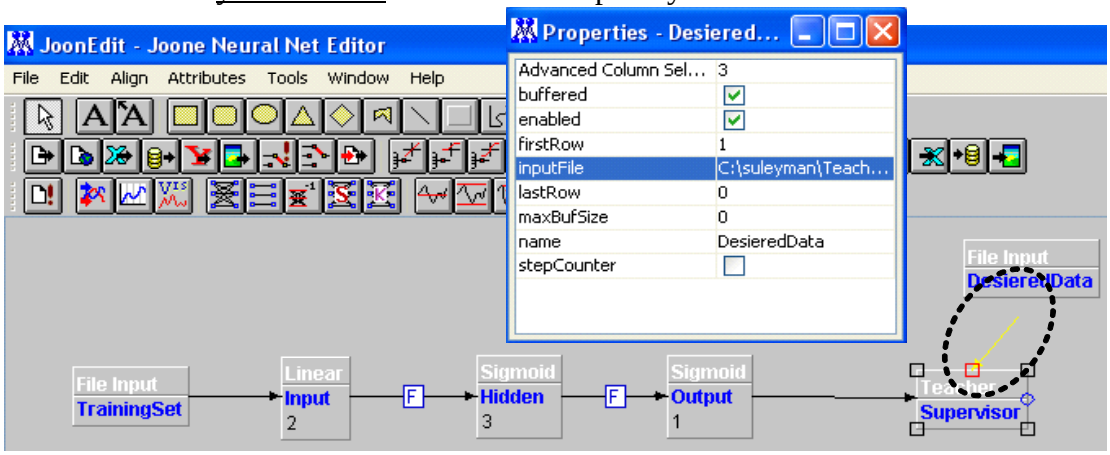
6. In order to train the neural network, a training set is provided by means of a file input layer. In our simple example, the first two columns of all rows are used. For that reason set the parameter 'Advanced Column Selector' to "1,2" or "1-2". Selecting 'firstRow' as 1 and 'lastRow' as 0 will force the usage of all rows in the text file that is specified in the field 'inputFile' (use 'binaryXOR_truth_table.txt'). Connect the input file to the input layer.



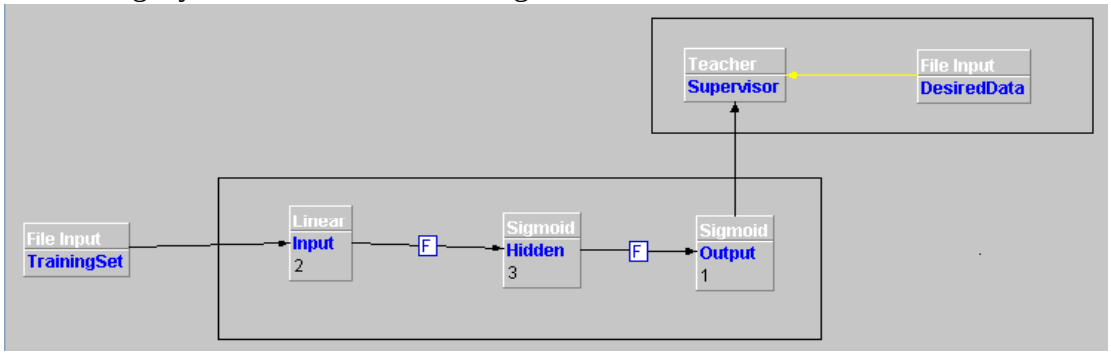
7. As the neural network is *supervised*, we need a *teacher*. Connect the output layer to the Teacher Layer (change the name to 'supervisor')



8. The ‘Supervisor’ must have access to the desired output for each pair of inputs that are sent to the network. Create another File Input layer and call it ‘DesiredData’. Set the different properties as shown below. Connect the ‘Supervisor’ to the ‘DesiredData’ by dragging a line from the little red square on the top side of the Teacher layer and then releasing the mouse button when the yellow arrow is on the File Input layer.

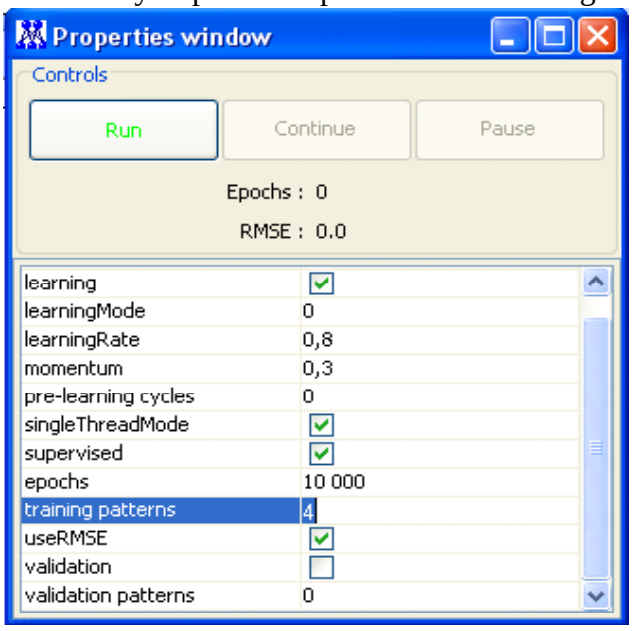


9. At this stage, you should have something similar to:



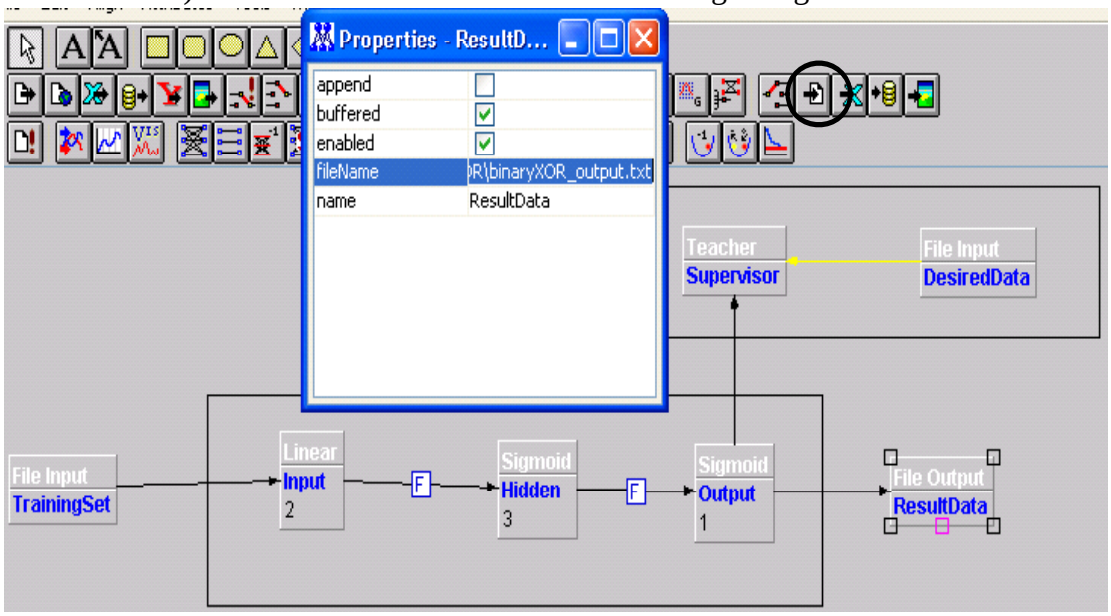
10. Now we need to teach the network how to solve the XOR problem. In the menu line, click on ‘Tools -> Control Panel’. Fill in the parameters as shown below. The parameter ‘training patterns’ is the number of rows in the training set. The entire set is sent to the network 10000 times (epochs). Click the ‘Run’ button to start the training procedure. The Control Panel shows the number of performed epochs and the current error. The final value should be less than ‘0.1’. If this is not the case, click on ‘Tools -> Randomize’ and ‘Tools -> Add

noise' in the menu line. This will randomize and add noise to the weights of the synapses and thereby improve the procedure of learning. Click 'Run' again!

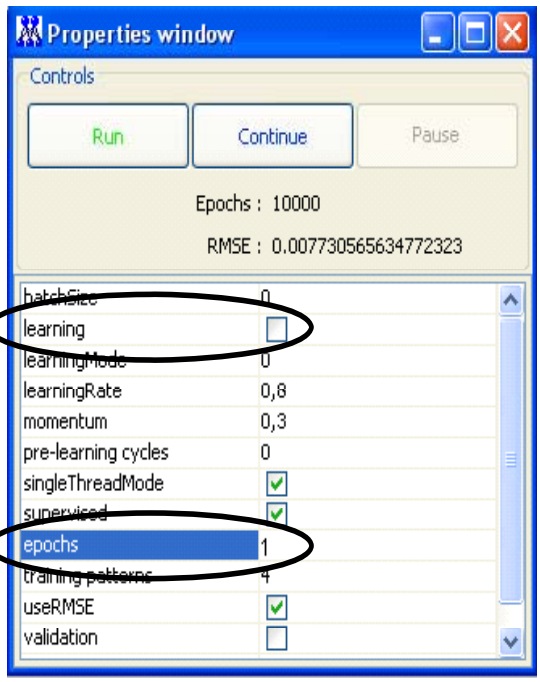


Testing the trained network

11. In order to test the trained XOR-network, add an Output File layer. In the 'Properties window', set the 'name' to ResultData and the 'fileName' to 'binaryXOR_output.txt' (including the path). When it comes to the Teacher layer, two options are possible: either it is kept connected to the network (together with the corresponding File Input, i.e. 'DesiredData') or it is removed. In both cases the testing will give same result!



12. Open the Control Panel, disable the 'learningRate' parameter and set the number of epochs to 1. By clicking on 'Run', a text file with name 'binaryXOR_output.txt' is created in your working directory.



13. The output file contains four values corresponding to the outputs in the truth table. The content should be similar to:

```
0.007830879673053221
0.9904490706938025
0.9903916946908758
0.013067862923140524
```

Hints:

- Tools ->Randomize: reset the weights of a neural network initializing it.
- Tools->Add Noise: random noise is added to the weights in order to permit the net to exit from a local minimum.
- If the network seems to "memorize" the training patterns from a previous training set, though a new training set is used, reset the input stream (Tools -> Reset Input Stream).
- It is possible to manually initialize synapse weights to certain values:
  - In a text editor, write the weight values using ';' as column separator (similar to the input file)
  - Copy the inserted values.
  - Inspect the synapse connection that needs to be initialized and press the 'paste' button.
- If the network needs to be retrained, disable the File Output layer 'ResultData'. This will eliminate the OutOfMemory error that is raised due to the limited java heap size. The heap is rapidly filled because of updating the output file 'binaryXOR_output.txt' as many time as epochs are specified!
- To test a trained network, you may need to save the network and re-open it!!
- The input layer of the XOR (binary / continuous) should use linear transfer function (not sigmoid). Otherwise, the parity neural network will not be trainable!

Appendix B.

The Parity Problem

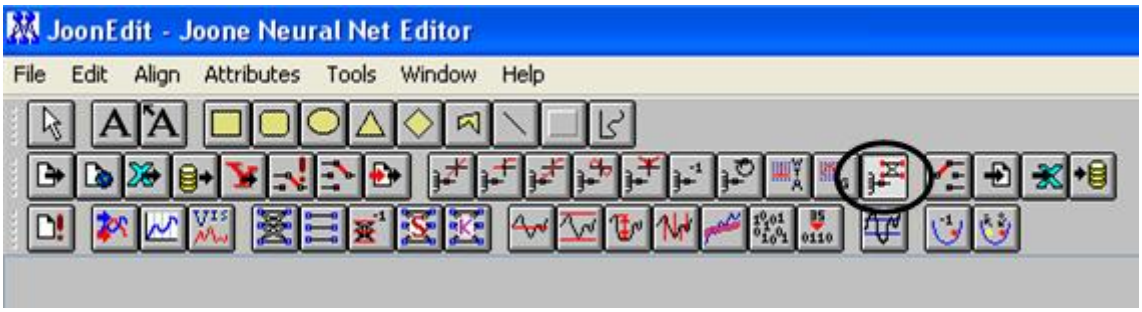
The parity problem has a long history in the study of neural networks. The *N*-bit parity function is a mapping defined on 2^N distinct binary vectors that indicates whether the sum of the N components of a binary vector is odd or even. In other words, the result of the mapping is 0 if the number of ones is even, and 1 otherwise. The truth table of 4-bit parity function, i.e. *N*=4, is given in as follows:

I_1	I_2	I_3	I_4	f
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	0
0	1	0	0	1
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	1
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	0

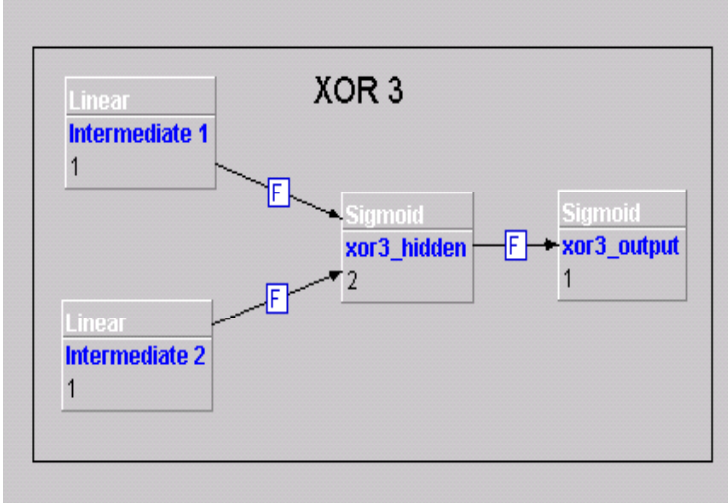
Many solution proposals make use of a standard Feedforward Neural Network. The most common used network architecture uses one input layer, one output layer and one hidden layer in between. The transfer function in both hidden and output layers is the sigmoid function. Such architectures require *N* nodes in the hidden layer to solve *N*-bit parity problem. In spite of the very long time the training procedure takes, the network may not learn to solve the problem! In this sense, modularity of neural networks provides a powerful solution. Actually, a better solution to the parity problem is obtained by a modular neural network composed of three instances of the XOR neural network presented before. Here, the output nodes of the first two XOR networks serve as an input layer to the third XOR network. Your task is to build a neural network (*BinaryParityNN*) that is trained to solve a 4-bit parity problem according to the truth table above.

In the following, a step-by-step manual will help you to build your *BinaryParityNN*.

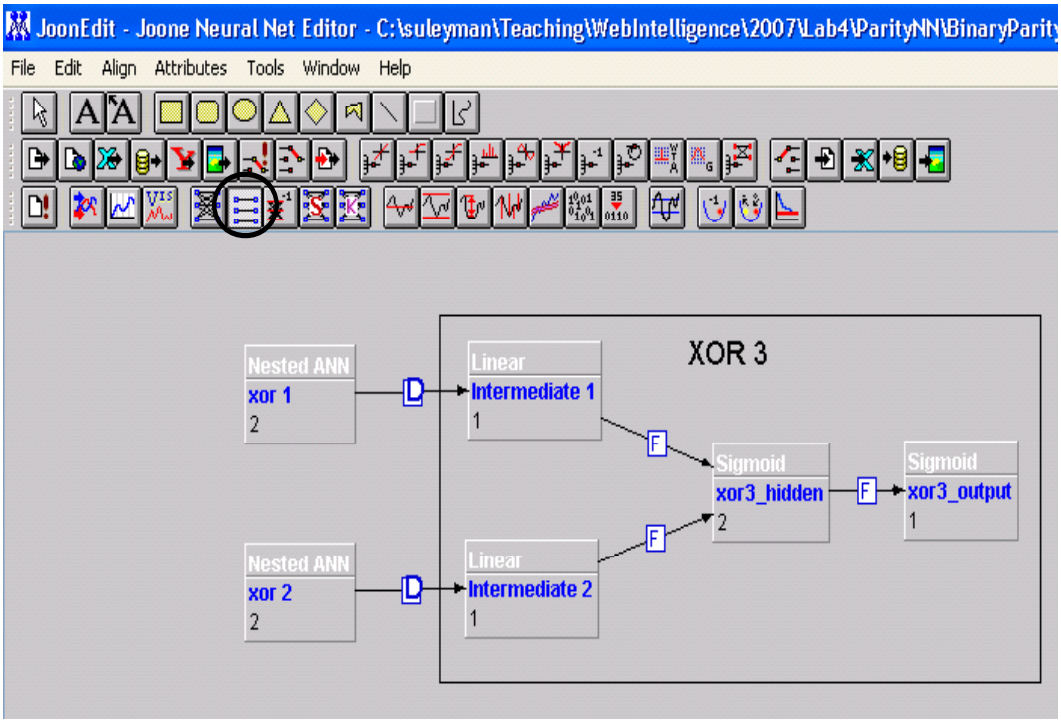
1. Before you start building the *BinaryParityNN*, you must save the XOR network in a form that can be inserted as a *NeuralNet* Object. Simply remove the teacher and all I/O components from your XOR network, before you save it in a serialized form. In the GUI Editor, choose File -> Export NeuralNet, and save it as “BinaryXOR.snet”.
2. The truth table is to be saved in a text file called “BinaryParity_truth_table.txt”.
3. In GUI Editor choose to build a new neural network.
4. Add two instances of the XOR neural networks by clicking the button for “New Nested NN”.



5. In the properties for both instances set the learning parameter to False (default) and link the Nested ANN to file “xor.snet”. Name the instances preferably as “xor 1” and “xor 2”.
6. As the input layer of the third XOR is composed of the outputs of the first two XORs, two different Linear layers serve as input layer to the third XOR. The hidden and the output layer use the sigmoid function as before. Add two Linear layers, call them “Intermediate 1” and “Intermediate 2”, with one node each (corresponding to the different outputs of “xor 1” and “xor 2”). Let the value of beta (in the properties) be 1.0 (default).
7. Now, add a hidden and the output layer, both of kind sigmoid. Call them “xor3_hidden” and “xor3_output” respectively. The hidden layer consists of two nodes and the output layer of 1 node only.
8. Connect the three layers of XOR 3 by using the Full Synapse. To ease the understanding of the diagram we group all the layers of XOR 3 together by drawing a rectangle.



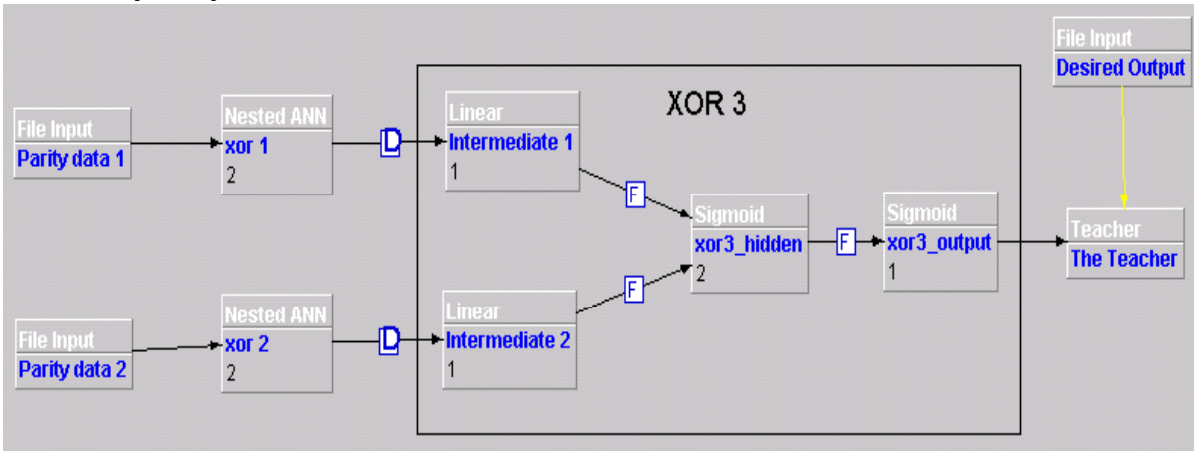
9. The architecture of the parity network “ParityNN” is competed by combining the three XORs. As the output from ‘xor 1’ serves as input to XOR 3, a direct connection between the networks is needed. Use the Direct Synapse to connect the networks as shown below.



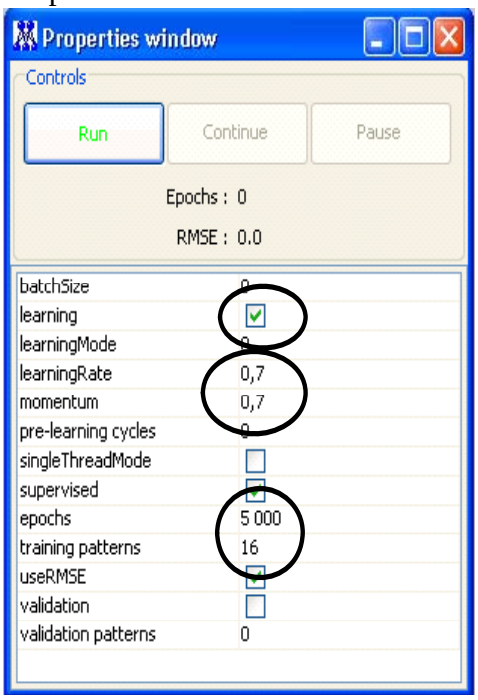
10. The modular network "ParityNN " is fed by input data through two File Input components, called "Parity data 1" and "Parity data 2".
11. Some of the properties of the File Input components are to be set according to the following table. All other default properties remain unchanged.

name	Parity data 1	Parity data 2
Advanced Column Selector	1-2	3-4
fileName	BinaryParity_truth_table.txt	BinaryParity_truth_table.txt
stepCounter	True	False

12. In order to train the network to solve the parity problem, a teacher is needed. Add a Teacher component and provide it with the desired output through a new File Input, called "Desired output", with the Advanced Column Selector set to 5. The property *fileName* is set to "BinaryParity_truth_table.txt".



13. Now is the ParityNN ready to be trained. Open the Control Panel (Tools -> Control Panel) and set the parameters as shown below. By running the network, a gradual descending RMSE value is observed, which shows that the network is learning the solution of the parity problem.



14. To verify the correctness of functionality, add a File Output component, call it “results”, and connect it to the output layer of XOR 3. Run the Control Panel again for **one** epoch only. Don’t forget to set the parameter of learning to False. The values in the obtained output file must agree with truth table of the parity function.